How to Make Your Own C++ Game Engine

Gustavo Pezzi

(this article is reposted from pikuma.com)

Game development has always been a great helper to get my students motivated to learn more about more advanced computer science topics.

One of my tutors, Dr. Sepi, once said:

> "Some people think games are kid's stuff, but gamedev is one of the few areas that uses almost every item of the standard CS curriculum."
> - Dr. Sepideh Chakaveh

She's absolutely right! If we expose what is hidden under the development stack of any modern game, we'll see that it touches many concepts that are familiar to any computer science student.

Algorithms & Data structures
Game programmers need to represent game objects in memory and access their data with performance in mind (data structures & data locality).
Mathematics
Game development can require many areas of applied math. Many games take advantage of discrete math, linear algebra, calculus, probability, numerical analysis, and many other branches of mathematics.
Computer architecture
Engine developers need a good understanding of the machine architecture they are targeting. This enables them to get metal speed.
Graphics
Graphics technologies are crucial to game development. Shaders, vertices, parallel acceleration, 2D & 3D rendering, and GPU APIs are some examples of graphics tech that developers use.
Compilers & formal languages
Engines often need to expose scripting languages to level designers, which usually means interpreting some sort of high-level custom syntax.
Artificial Intelligence
Game objects (enemies and NPCs) often need to portrait human-like intelligence. Some examples are path-finding algorithms, chasing, machine learning, and visibility methods.
Networks
Multiplayer and social integration are common in modern games. Most games must take advantage of machine connectivity and send/receive multiple packets via network procotols.
UI & UX
User interfaces and UX theory is also a huge part of gamedev tech. This means displaying information to and providing appropriate feedback to the users.
pikuma.com

Depending on the nature of your game, you might need to dive even deeper into more specialized areas, like distributed systems or human-computer interaction. Game development is serious business and it can be a powerful tool to learn serious CS concepts.

This article will go over some of the fundamental building blocks that are required to create a simple game engine with C++. I'll explain the main elements that are required in a game engine, and give some personal recommendations on how I like to approach writing one from scratch.

That being said, this will **not** be a coding tutorial. I won't go into too much technical detail or explain how all these elements are glued together via code. If you are looking for a comprehensive video book on how to write a C++ game engine, this is a great starting point: Create a 2D Game Engine with C++ & Lua.

What is a Game Engine?

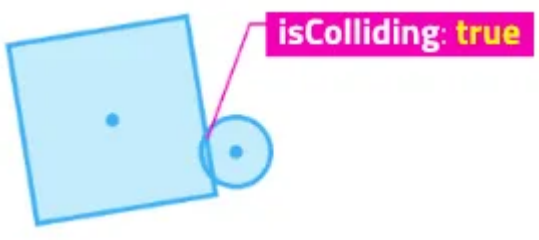
If you are reading this, chances are you already have a good idea of what a game engine is, and possibly even tried to use one yourself. But just so we are all on the same page, let's quickly review what game engines are and what they help us achieve.

A **game engine** is a set of software tools that optimizes the development of video games. These engines can be small and minimalist, providing just a game loop and a couple of rendering functions, or be large and comprehensive, similar to IDE-like applications where developers can script, debug, customize level logic, AI, design, publish, collaborate, and ultimately build a game from start to finish without the need to ever leave the engine.

Game engines and game frameworks usually expose an API to the user. This API allows the programmer to call engine functions and perform hard tasks as if they were black boxes.

To really understand how this API thing works, let's put it into context. For example, it is not rare for a game engine API to expose a function called "**IsColliding()**" that developers can invoke to check if two game objects are colliding or not. There is no need for the programmer to know **how** this function is implemented or what is the algorithm required to correctly determine if two shapes are overlapping. As far as we are concerned, the **IsColliding** function is a black box that does some magic and correctly returns *true* or *false* if those objects are colliding with each other or not. This is an example of a function that most game engines expose to their users.

```
if (IsColliding(player, bullet)) {
  lives--;
  if (lives == 0) {
    GameOver();
  }
}
```



Besides a programming API, another big responsibility of a game engine is hardware abstraction. For example, 3D engines are usually built upon a dedicated graphics API like OpenGL, Vulkan, or Direct3D. These APIs provide a software abstraction for the Graphics Processing Unit (GPU).

Speaking of hardware abstraction, there are also low-level libraries (like DirectX, OpenAL, and SDL) that provide abstraction & multi-platform access to many

other hardware elements. These libraries help us access and handle keyboard events, mouse movement, network connection, and even audio.

The Rise of Game Engines

In the early years of the game industry, games were built using a custom rendering engine and the code was developed to squeeze as much performance as possible from slower machines. Every CPU cycle was crucial, so code reuse or generic functions that worked for multiple scenarios was not a luxury that developers could afford.

As games and development teams grew in both size and complexity, most studios ended up reusing functions and subroutines between their games. Studios developed in-house engines that were basically a collection of internal files and libraries that dealt with low-level tasks. These functions allowed other members of the development team to focus on high-level details like gameplay, map creation, and level customization.

Some popular classic engines are id Tech, Build, and AGI. These engines were created to aid the development of specific games, and they allowed other members of the team to rapidly develop new levels, add custom assets, and customize maps on the fly. These custom engines were also used to mod or create expansion packs for their original games.

Id Software developed id Tech. id Tech is a collection of different engines where each iteration is associated with a different game. It is common to hear developers describe id Tech 0 as "the Wolfenstein3D engine", id Tech 1 as "the Doom engine", and id Tech 2 as "the Quake engine."

Build is another example of engine that helped shape the history of 90's games. It was created by Ken Silverman to help customize first-person shooters. Similar to what happened to id Tech, Build evolved with time and its different versions helped programmers develop games such as Duke Nukem 3D, Shadow Warrior, and Blood. These are arguably the most popular titles created using the Build engine, and are often referred as "The Big Three."

Yet another example of game engine from the 90s was the "Script Creation Utility for Manic Mansion" (SCUMM). SCUMM was an engine developed at *LucasArts*, and it is the base of many classic Point-and-Click games like Monkey Island and Full Throttle.

The SCUMM scripting language was used to manage Full Throttle's dialogs and actions.

As machines evolved and became more powerful, so did game engines. Modern engines are packed with feature-rich tools that require fast processor speeds, ridiculous amounts of memory, and dedicated graphics cards.

With power to spare, modern engines trade machine cycles for more abstraction. This trade-off means we can view modern game engines as general-purpose tools to create complex games at low cost and short development times.

Why Make a Game Engine?

This is a very common question, and different game programmers will have their own take on this topic depending on the nature of the game being developed, their business needs, and other driving forces being considered.

There are many free, powerful, and professional commercial engines that developers can use to create and deploy their own games. With so many game engines to choose from, why would anyone bother to make a game engine from the ground up?

I wrote a blog post explaining some of the reasons programmers might decide to make a game engine from scratch. In my opinion, the top reasons are:

Learning opportunity: a low-level understanding of how game engines work under the hood can make you grow as a developer.

Workflow control: you'll have more control over special aspects of your game and adjust the solution to fit your workflow needs.

Customization: you'll be able to tailor a solution for a unique game requirement.

Minimalism: a smaller codebase can reduce the overhead that comes with bigger game engines.

Innovation: you might need to implement something completely new or target an unorthodox hardware that no other engine supports.

I will continue our discussion assuming you are interested in the **educational** appeal of game engines. Creating a small game engine from scratch is something I strongly recommend to all my CS students.

How to Make a Game Engine

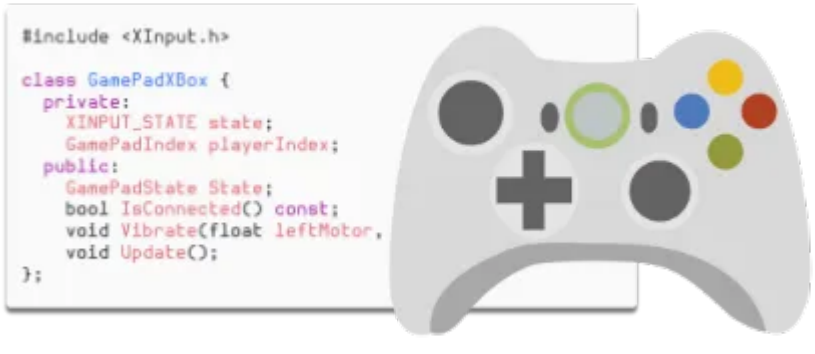
So, after this quick talk about the motivations of using and developing game engines, let's go ahead and discuss some of the components of game engines and learn how we can go about writing one ourselves.

1. Choosing a Programming Language

One of the first decisions we face is choosing the programming language we'll use to develop the core engine code. I have seen engines being developed in raw assembly, C, C++, and even high-level languages like C#, Java, Lua, and even JavaScript!

One of the most popular languages for writing game engines is C++. The C++ programming language combines speed with the ability to use object-oriented programming (OOP) and other programming paradigms that help developers organize and design large software projects.

Since performance is usually a great deal when we develop games, C++ has the advantage of being a compiled language. A compiled language means that the final executables will run natively on the processor of the target machine. There are also many dedicated C++ libraries and development kits for most modern consoles, like PlayStation or Xbox.



For example, developers can access the Xbox controller using C++ libraries provided by Microsoft itself.

Speaking of performance, I personally don't recommend languages that use virtual machines, bytecode, or any other intermediary layer. Besides C++, some modern alternatives that are suited for writing core game engine code are Rust, Odin, and Zig.

For the remainder of this article, my recommendations will assume the reader wants to build a simple game engine using the C++ programming language.

2. Hardware Access

In older operating systems, like the MS-DOS, we could usually poke memory addresses and access special locations that were mapped to different hardware components. For example, all I had to do to "paint" a pixel with a certain color was to load a special memory address with the number that represented the correct color of my VGA palette, and the display driver translated that change to the physical pixel into the CRT monitor.

As operating systems evolved, they became responsible for protecting the hardware from the programmer. Modern operating systems will not allow the code to modify memory locations that are outside the allowed addresses given to our process by the OS.

For example, if you are using Windows, macOS, Linux, or *BSD, you'll need to ask the OS for the correct permissions to draw and paint pixels on the screen or talk to any other hardware component. Even the simple task of opening a window on the OS desktop is something that must be performed via the operating system API.

Therefore, running a process, opening a window, rendering graphics on the screen, paining pixels inside that window, and even reading input events from the keyboard are all OS-specific tasks.

One very popular library that helps with multi-platform hardware abstraction is SDL. I personally like using SDL when I teach gamedev classes because with SDL I don't need to create one version of my code for Windows, another version for macOS, and another one for Linux students. SDL works as a bridge not just for different operating systems, but also different CPU architectures (Intel, ARM, Apple M1, etc.). The SDL library abstracts the low-level hardware access and "translates" our code to work correctly on these different platforms.

Here is a minimal snippet of code that uses SDL to open a window on the operating system. I'm not handling errors for the sake of simplicity, but the code below will be the same for Windows, macOS, Linux, BSD, and even RaspberryPi.

```
#include <SDL2/SDL.h>
void OpenNewWindow() {
  SDL_Init(SDL_INIT_VIDEO);
  SDL_Window* window = SDL_CreateWindow("My Window", 0, 0, 800, 600, 0);
  SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);
}
```

But SDL is just one example of library that we can use to achieve this multi-platform hardware access. SDL is a popular choice for 2D games and to port existing code to different platforms and consoles. Another popular option of multi-platform library that is used mostly with 3D games and 3D engines is GLFW. The GLFW library communicates very well with accelerated 3D APIs like OpenGL and Vulkan.

3. Game Loop

Once we have our OS window open, we need to create a controlled game loop.

Put simply, we *usually* want our games to run at 60 frames per second. The framerate might be different depending on the game, but to put things into perspective, movies shot on film run at a 24 FPS rate (24 images flash past your eyes every single second).

A game loop **runs continuously during gameplay**, and at each pass of the loop, our engine needs to run some important tasks. A traditional game loop must:

Process Input events without blocking

Update all game objects and their properties for the current frame

Render all game objects and other important information on the screen

```
while (isRunning) {
  Input();
  Update();
  Render();
}
```

That's a cute while-loop. Are we done? Absolutely not!

A raw C++ loop is not good enough for us. A game loop must have some sort of relationship with real-world time. After all, the enemies of the game should move at the same speed on a any machine, regardless of their CPU clock speed.

Controlling this framerate and setting it to a fixed number of FPS is actually a very interesting problem. It usually requires us to keep track of time between frames and perform some reasonable calculations to make sure our games run smoothly at a framerate of at least 30 FPS.

4. Input

I cannot imagine a game that does not read some sort of input event from the user. These can come from a keyboard, a mouse, a gamepad, or a VR set. Therefore, we must process and handle different input events inside our game loop.

To process user input, we must request access to hardware events, and this must be performed via the operating system API. The good news is that we can use a multi-platform hardware abstraction library (SDL, GLFW, SFML, etc.) to handle user input for us.

If we are using SDL, we can poll events and proceed to handle them accordingly with a few lines of code.

```
void Input() {
  SDL_Event event;
  while (SDL_PollEvent(&event)) {
    switch (event.type) {
      case SDL_KEYDOWN:
        if (event.key.keysym.sym == SDLK_SPACE) {
          ShootMissile();
        }
        break;
    }
  }
}
```

Once again, if we are using a cross-platform library like SDL to handle input, we don't have to worry too much about OS-specific implementation. Our C++ code should be the same regardless of the platform we are targeting.

After we have a working game loop and a way of handling user input, it's time for us to start thinking about organizing our game objects in memory.

5. Representing Game Objects in Memory

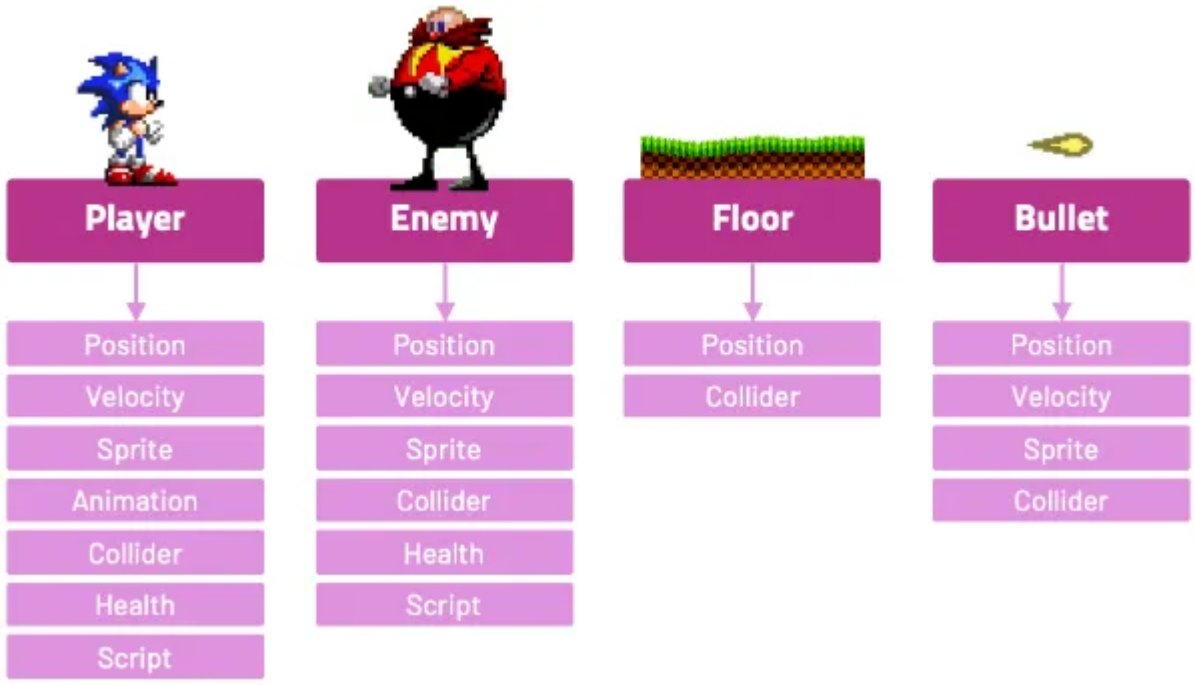
When we are designing a game engine, we need to setup data structures to store and access the objects of our game.

There are several techniques that programmers use when architecturing a game engine. Some engines might use a simple object-oriented approach with classes and inheritance, while other engines might organize their objects as entities and components.

If one of your goals is to learn more about algorithms and data structures, my recommendation is for you to try implementing these data structures yourself. If you're using C++, one option is to use the STL (standard template library) and take advantage of the many data structures that come with it (vectors, lists, queues, stacks, maps, sets, etc.). The C++ STL relies heavily on templates, so this can be a good opportunity to practice working with templates and see them in action in a real project.

As you start reading more about game engine architecture, you'll see that one of the most popular design patterns used by games is based on **entities** and **components**. An entity-component design organizes the objects of our game scene as *entities* (what Unity calls "game objects" and Unreal calls "actors"), and *components* (the data that we can add or attach to our entities).

To understand how entities and components work together, think of a simple game scene. The entities will be our main player, the enemies, the floor, the projectiles, and the components will be the important blocks of data that we "attach" to our entities, like position, velocity, rigid body collider, etc.



A popular game engine design patterns is to organize game elements as entities and components.

Some examples of components that we can choose to attach to our entities are:

Position component: Keeps track of the x-y position coordinates of our entity in the world (or x-y-z in 3D).

Velocity component: Keeps track of how fast the entity is moving in the x-y axis (or x-y-z in 3D).

Sprite component: It usually stores the PNG image that we should render for a certain entity.

Animation component: Keeps track of the entity's animation speed, and how the animation frames change over time.

Collider component: This is usually related to physics characteristics of a rigid body, and defines the colliding shape of an entity (bounding box, bounding circle, mesh collider, etc.).

Health component: Stores the current health value of an entity. This is usually just a number or in some cases a percentage value (a health bar, for example).

Script component: Sometimes we can have a script component attached to our entity, which might be an external script file (Lua, Python, etc) that our engines must interpret and execute behind the scenes.

This is a very popular way of representing game objects and important game data. We have entities, and we "plug" different components to our entities.

There are many books and articles that explore *how* we should go about implementing an entity-component design, as well as what data structures we should use in this implementation. The data structures we use and how we access them have a direct impact on our game's performance, and you'll hear developers mention things like Data-Oriented Design, Entity-Component-System (ECS), data locality, and many other ideas that have everything to do with how our game data is stored in memory and how we can access this data efficiently.

Representing and accessing game objects in memory can be a complex topic. In my opinion, you can either code a simple entity-component implementation manually, or you can simply use an existing third-party ECS library.

There are some popular options of ready-to-use ECS libraries that we can include in our C++ project and start creating entities and attaching components without having to worry about *how* they are implemented under the hood. Some examples of C++ ECS libraries are EnTT and Flecs.

I personally recommend students that are serious about programming to try implementing a very simple ECS manually at least once. Even if your

implementation is not perfect, coding an ECS system from scratch will force you to think about the underlying data structures and consider their performance.

Now, serious talk! Once you're done with your custom ad-hoc ECS implementation, I would encourage you to just use some of the popular third-party ECS libraries (EnTT, Flecs, etc.). These are professional libraries that have been developed and tested by the industry for several years. They are probably a lot better than anything we could come up from scratch ourselves.

In summary, a professional ECS is difficult to implement from scratch. It is valid as an academic exercise, but once you're done with your small learning project, just pick a well-tested third-party ECS library and add it to your game engine code.

6. Rendering

Alright, it looks like our game engine is slowly growing in complexity. Now that we have discussed about ways of storing and accessing game objects in memory, we need to probably talk about how we render objects on the screen.

The first step is to consider the nature of the games that we will be creating with our engine. Are we creating a game engine to develop only 2D games? If that's the case, we need to think about rendering sprites, textures, managing layers, and probably take advantage of graphics card acceleration. The good news is that 2D games are usually simpler than 3D ones, and 2D math is considerably easier than 3D math.

If your goal is to develop a **2D** engine, you can use SDL to help with multi-platform rendering. SDL abstracts accelerated GPU hardware, can decode and display PNG images, draw sprites, and render textures inside our game window.

Now, if your goal is to develop a **3D** engine, then we'll need to define how we send some extra 3D information (vertices, textures, shaders, etc) to the GPU. You'll probably want to use a software abstraction to the graphics hardware, and the most popular options are OpenGL, Direct3D, Vulkan, and Metal. The decision of which API to use might depend on your target platform. For example, Direct3D will power Microsoft apps, while Metal will work solely with Apple products.

3D applications work by processing 3D data through a graphics pipeline. This pipeline will dictate how your engine must send graphics information to the GPU (vertices, texture coordinates, normals, etc.). The graphics API and the pipeline will also dictate how we should write programmable shaders to transform and modify vertices and pixels of our 3D scene.

Programmable shaders dictate how the GPU should process and display 3D objects. We can have different scripts per vertex and per pixel (fragment), and they control reflection, smoothness, color, transparency, etc.

Speaking of 3D objects and vertices, it is a good idea to delegate to a library the task of reading and decoding different mesh formats. There are many popular 3D model formats that most third-party 3D engines should be aware of. Some examples of files are .OBJ, Collada, FBX, and DAE. My recommendation is to start with .OBJ files. There are well-tested and well-supported libraries that handle OBJ loading with C++. TinyOBJLoader and AssImp are great options that are used by many game engines.

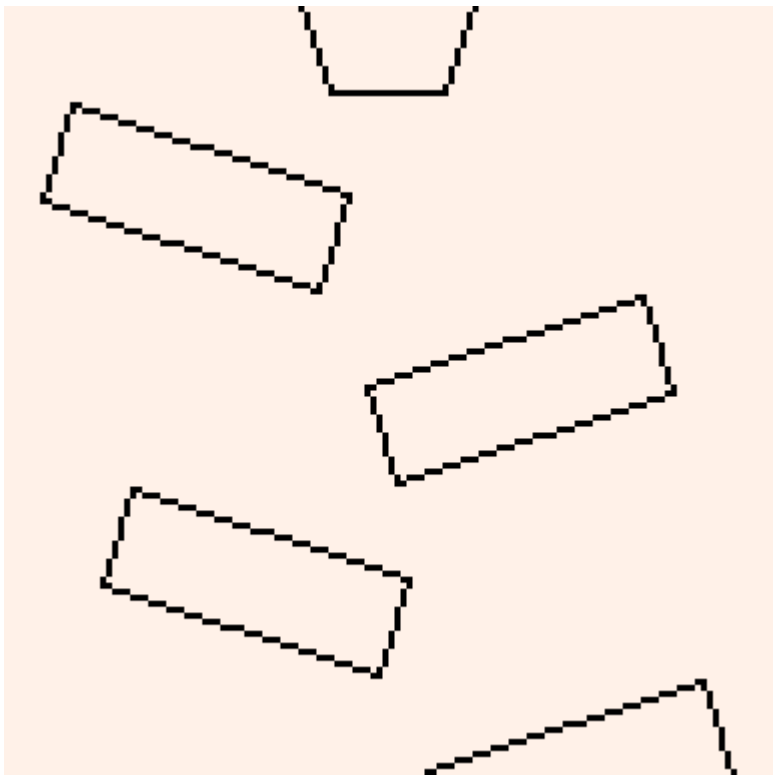
7. Physics

When we add entities to our engine, we probably also want them to move, rotate, and bounce around our scene. This subsystem of a game engine is the physics simulation. This can either be created manually, or imported from an existing ready-to-use physics engine.

Here, we also need to consider what type of physics we want to simulate. 2D physics is usually simpler than 3D, but the underlaying parts of a physics simulation are very similar to both 2D and 3D engines.

If you simply want to include a physics library to your project, there are several great options to choose from.

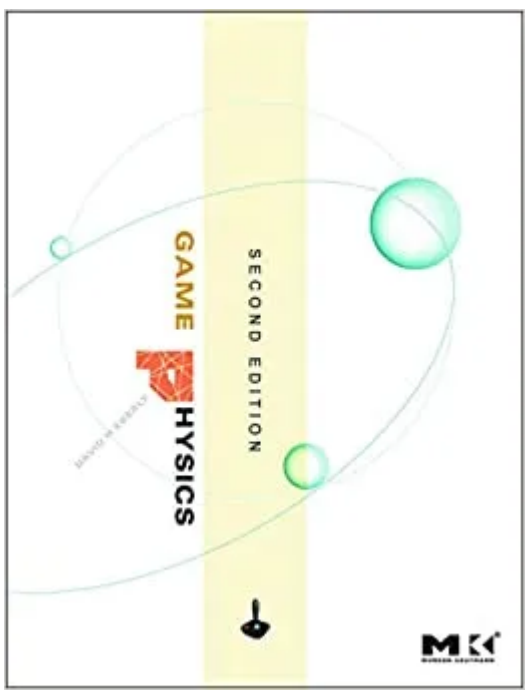
For 2D physics, I recommend looking at Box2D and Chipmunk2D. For professional and stable 3D physics simulation, some good names are libraries like PhysX and Bullet. Using a third-party physics engine is always a good call if physics stability and development speed are crucial for your project.



As an educator, I strongly believe every programmer should learn how to code a simple physics engine at least once in their career. Once again, you don't need to write a perfect physics simulation, but focus on making sure objects can accelerate correctly and that different types of forces can be applied to your game objects. And once movement is done, you can also think of implementing some simple collision detection and collision resolution.

If you want to learn more about physics engines, there are some good books and online resources that you can use. For 2D rigid-body physics, you can look at the Box2D source code and the slides from Erin Catto. But if you are looking for a comprehensive course about game physics, 2D Game Physics from Scratch is probably a good place to start.

If you want to learn about 3D physics and how to implement a robust physics simulation, another great resource is the book "Game Physics" by David Eberly.



8. UI

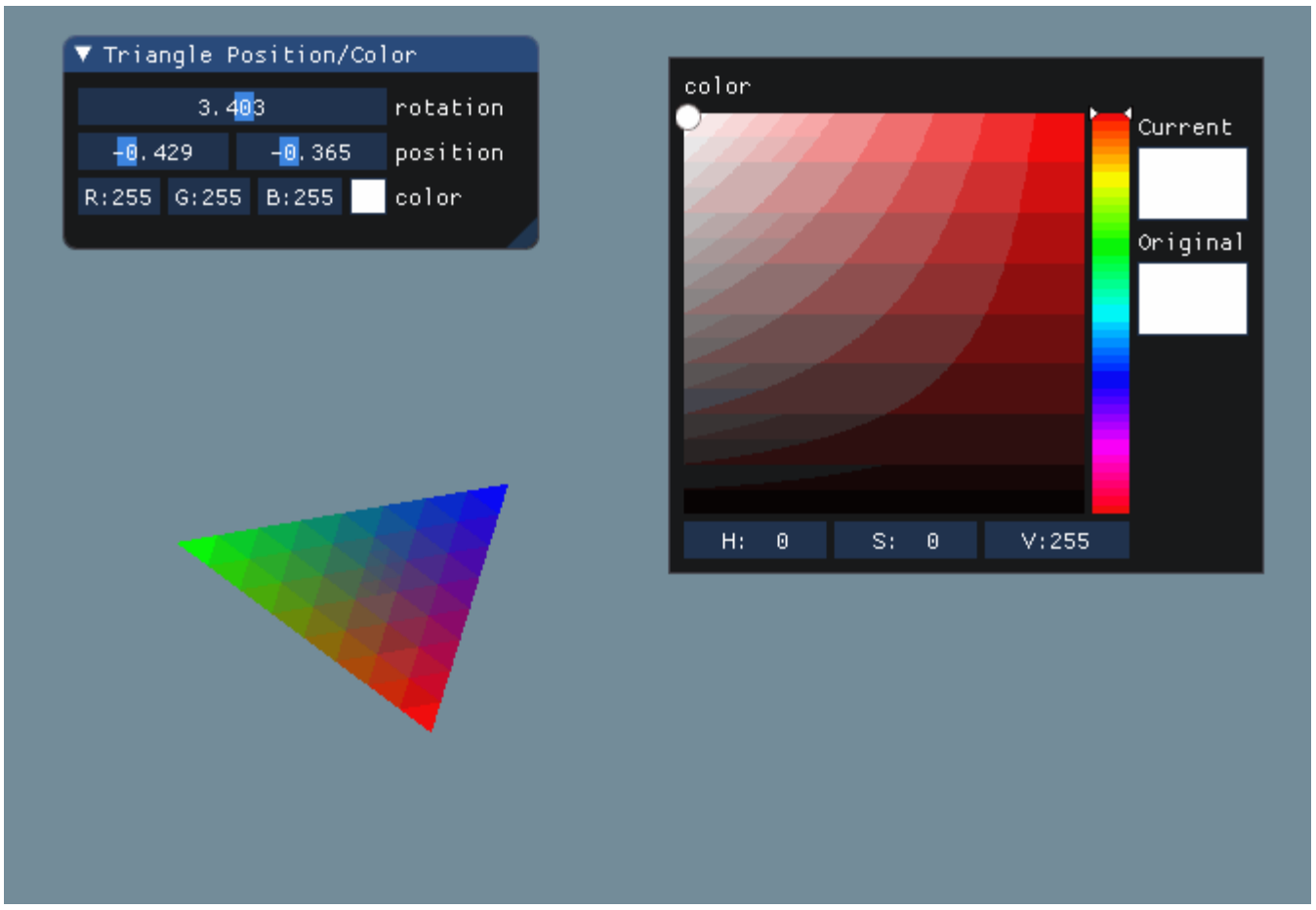
When we think of modern game engines like Unity or Unreal, we think of complex user interfaces with many panels, sliders, drag-and-drop options, and other pretty UI elements that help users customize our game scene. The UI allows the developer to add and remove entities, change component values on-the-fly, and easily modify game variables.

Just to be clear, we are talking about game engine UI for tooling, and not the user interface that we show to the users of your game (like dialog screens and menus).

Keep in mind that game engines do not necessarily need to have an editor embedded to them, but since game engines are usually used to increase productivity, having a friendly user interface will help you and other team members to rapidly customize levels and other aspects of the game scene.

Developing a UI framework from the ground up is probably one of the most annoying tasks that a beginner programmer can attempt to add to a game engine. You'll have to create buttons, panels, dialog boxes, sliders, radio buttons, manage colors, and you'll also need to correctly handle the events of that UI and always persist its state. Not fun! Adding UI tools to your engine will make your application increase in complexity and add an incredible amount of noise to your source code.

If your goal is to create UI tools for your engine, my recommendation is to use an existing third-party UI library. A quick Google search will show you that the most popular options are Dear ImGui, Qt, and Nuklear.



Dear ImGui is one of my favorites, as it allows us to quickly setup user interfaces for engine tooling. The ImGui project uses a design pattern called "immediate mode UI", and it is widely used with game engines because it communicates well with 3D applications by taking advantage of accelerated GPU rendering.

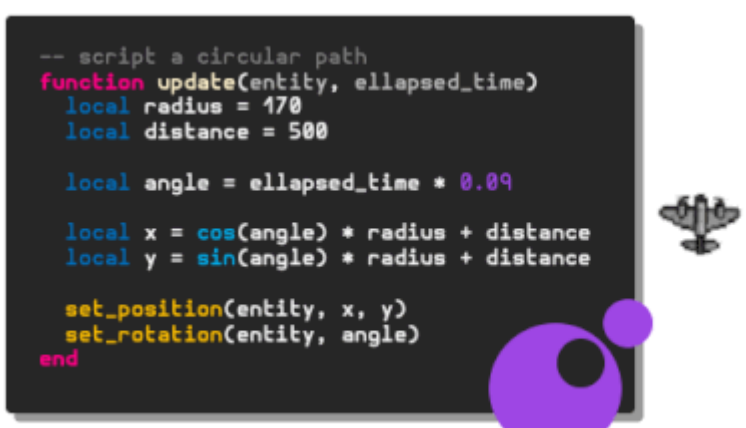
In summary, if you want to add UI tools to your game engine, my suggestion is to simply use Dear ImGui.

9. Scripting

As our game engine grows, one popular option is to enable level customization using a simple scripting language.

The idea is simple; we embed a scripting language to our native C++ application, and this simpler scripting language can be used by non-professional programmers to script entity behavior, AI logic, animation, and other important aspects of our game.

Some of the popular scripting languages for games are Lua, Wren, C#, Python, and JavaScript. All these languages operate at a considerably higher-level than our native C++ code. Whoever is scripting game behavior using the scripting language does *not* need to worry about things like memory management or other low-level details of how the core engine works. All they need to do is script the levels and our engine knows how to interpret the scripts and perform the hard tasks behind the scenes.


-- script a circular path
function update(entity, ellapsed_time)
local radius = 170
local distance = 500
local angle = ellapsed_time * 0.09
local x = cos(angle) * radius + distance
local y = sin(angle) * radius + distance
set_position(entity, x, y)
set_rotation(entity, angle)
end

My favorite scripting language is Lua. Lua is small, fast, and extremely easy to integrate with C and C++ native code. Also, if I'm working with Lua and "modern" C++, I like to use a wrapper library called Sol. The Sol library helps me hit the ground running with Lua and offers many helper functions to improve the traditional Lua C-API.

If we enable scripting, we are almost reaching a point where we can start talking about more advanced topics in our game engine. Scripting helps us define AI logic, customize animation frames and movement, and other game behavior that does **not** need to live inside our native C++ code and can easily be managed via external scripts.

10. Audio

Another element that you might consider adding support to a game engine is audio.

It is no surprise that, once again, if we want to poke audio values and emit sound, we need to access audio devices via the OS. And once again, since we don't *usually* want to write OS-specific code, I am going to recommend using a multi-platform library that abstracts audio hardware access.

Multi-platform libraries like SDL have extensions that can help your engine handle things like music and sound effects.

But, serious talk now! I would strongly suggest tackling audio only after you have the other parts of your engine already working together. Emitting sound files

can be easy to achieve, but once we start dealing with audio synchronization, linking audio with animations, events, and other game elements, things can become messy.

If you are really doing things manually, audio can be tricky because of multi-threading management. It can be done, but if your goal is to write a simple game engine, this is one part that I like to delegate to a specialized library

Some good libraries and tools for audio that you can consider integrating with your game engine are SDL_Mixer, SoLoud, and FMOD.

Tiny Combat Arena is an example of a game that uses the FMOD library for audio effects like doppler and compression. You can hear the afterburners and the 3D effects of the other jets passing by.

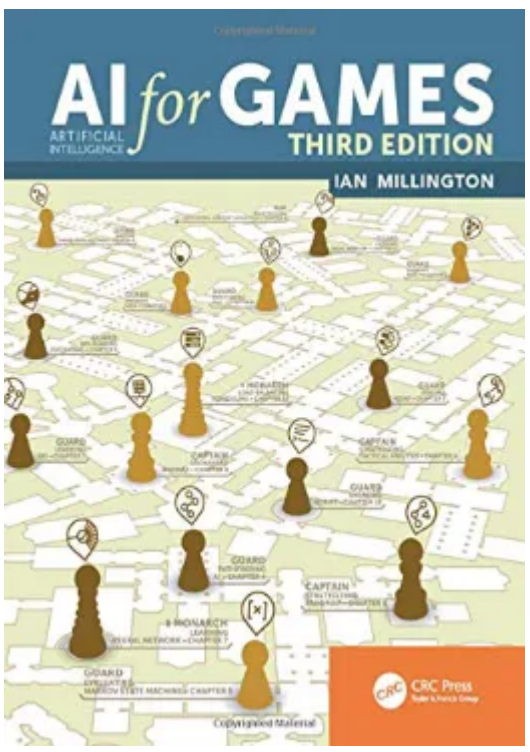
11. Artificial Intelligence

The final subsystem I'll include in our discussion is AI. We could achieve AI via scripting, which means we could delegate the AI logic to level designers to script. Another option would be to have a proper AI system embedded to our game engine core native code.

In games, AI is used to generate responsive, adaptive, or intelligent-like behavior to game objects. Most AI logic is added to non-player characters (NPCs, enemies) to simulate human-like intelligence.

Enemies are a popular example of AI application in games. Game engines can create abstractions over path-finding algorithms or interesting human-like behavior when enemies chase objects on a map.

A comprehensive book about the theory and implementation of artificial intelligence for games is called AI for Games by Ian Millington.



Don't Try to do Everything at Once

Alright! We have just discussed some important ideas that you can consider adding to a simple C++ game engine. But before we start gluing all these pieces together, I just want to mention something super important.

One of the hardest parts of working on a game engine is that most developers will not set clear boundaries, and there is no sense of "end line." In other words, programmers will start a game engine project, render objects, add entities, add components, and it's all downhill after that. If they don't define boundaries, it's easy to just start adding more and more features and lose track of the big picture. If that happens, there is a big chance that the game engine will never see the light of day.

Besides lacking boundaries, it is easy to get overwhelmed as we see the code grow in front of our eyes at lightning speed. A game engine project has the potential of quickly growing in complexity and in a few weeks, your C++ project can have several dependencies, require a complex build system, and the overall readability of your code drops as more features are added to the engine

One of my first suggestions here is to always write your game engine **while** you're writing an actual game. Start and finish the first iteration of your game with an actual game in mind. This will help you to set limits and define a clear path for what you needs to be completed. Try your best to stick to it and not get tempted to change the requirements along the way.

Take Your Time & Focus on the Basics

If you’re creating your own game engine as a learning exercise, enjoy the small victories!

Most student gets super excited at the beginning of the project, and as days go by the anxiety starts to appear. If we are creating a game engine from scratch, especially when using a complex language like C++, it is easy to get overwhelmed and lose some momentum.

I want to encourage you to fight that feeling of "running against time." Take a deep breath and enjoy the small wins. For example, when you learn how to successfully display a PNG texture on the screen, savor that moment and make sure you understand what you did. If you managed to successfully detect the collision between two bodies, enjoy that moment and reflect on the knowledge that you just gained.

Focus on the fundamentals and **own** that knowledge. It doesn't matter how small or simple a concept is, **own it**!!! Everything else is ego.

-->-->-->-->